Lecture 8

Game Programming Patterns

Assoc. Prof. Dr. Burkay Genç

02 May, 2024

Observer Pattern

Observer Pattern

- Consider an achievement system
  - Many badges to earn
  - Each achievement has its own requirements to be completed
    - Score 100 points
    - Collect 500 gold
    - Kill 10 enemies
    - Play for 2 hours

Bad Way To Do This

- We could simply check for these requirements where they happened
  - And then we could call the achievement functions to open achievements
- This is a terrible way to do this
  - Achievement related code will be spread to all over the game code
  - It will make reading other parts of the code difficult
  - It will make it difficult to detect where each achievement code is “hidden”
  - It will make it difficult to alter the achievement code
    - Other parts of the code may depend on it

Good Way To Do This

- The better way to do this is to use the *observer* pattern
- When an interesting thing happens
  - Raise a notification to let anybody interested about this event be notified

```
void Physics::updateEntity(Entity& entity)
{
  bool wasOnSurface = entity.isOnSurface();
  entity.accelerate(GRAVITY);
  entity.update();
  if (wasOnSurface && !entity.isOnSurface())
  {
    notify(entity, EVENT_START_FALL);
  }
}
```

- With this code, you can simply remove the achievement code
  - And nobody is hurt

How It Works

- The Observer pattern has two sides
  - The Observer
    - Listens for interesting events
    - Receives notifications
  - The Subject
    - Produces interesting events
    - Produces notifications

Code - Observer

```
class Observer
{
public:
  virtual ~Observer() {}
  virtual void onNotify(const Entity& entity, Event event) = 0;
};
```

```
class Achievements : public Observer
{
public:
  virtual void onNotify(const Entity& entity, Event event) {
    switch (event)
    {
    case EVENT_ENTITY_FELL:
      if (entity.isHero() && heroIsOnBridge_)
      {
        unlock(ACHIEVEMENT_FELL_OFF_BRIDGE);
      }
      break;

      // Handle other events, and update heroIsOnBridge_...
    }
  }

private:
  void unlock(Achievement achievement)   {
    // Unlock if not already unlocked...
  }

  bool heroIsOnBridge_;
};
```

Code - Subject

```
class Subject
{

private:
  Observer* observers_[MAX_OBSERVERS];
  int numObservers_;

public:
  void addObserver(Observer* observer)
  {
    // Add to array...
  }

  void removeObserver(Observer* observer)
  {
    // Remove from array...
  }

protected:
  void notify(const Entity& entity, Event event)
  {
    for (int i = 0; i < numObservers_; i++)
    {
      observers_[i]->onNotify(entity, event);
    }
  }
};
```

Problems?

- Slow observers can block subjects
  - The observer must be quick to return control to the subject
  - Or simply move on to another thread
  - This is especially critical for the UI subjects
- Too much dynamic allocation
  - Observer list can be dynamically allocated
  - But that usually only changes at the start of the game
  - If you are going to change it much, you can use a linked list
- Deleting subjects or observers
  - Be careful deleting observers
    - You must also delete the corresponding list entry in the subject
  - Deleting subjects is less harmful
  - The observers now listen to a non-existing subject
  - Best remedy: give a dying breath notification

Prototype Pattern

Prototype Pattern

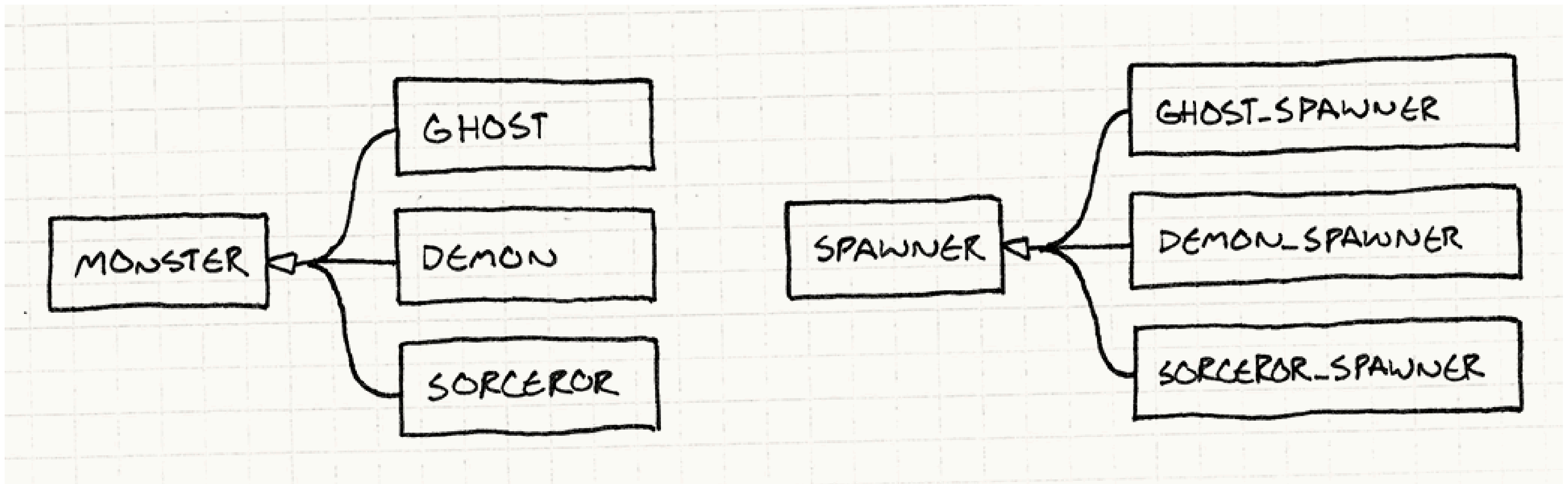
- Pretend we're making a game in the style of Crimson Land.
  - We've got creatures and fiends swarming around the hero,
  - They enter the arena through "spawners",
  - There is a different spawner for each kind of enemy.
- Let's say we have different classes for each kind of monster in the game
  - Ghost, Demon, Sorcerer, etc.

```
class Monster
{
  // Stuff...
};

class Ghost : public Monster {};
class Demon : public Monster {};
class Sorcerer : public Monster {};
```

Prototype Pattern

- A spawner constructs instances of one particular monster type.
- To support every monster in the game, we could have a spawner class for each monster class
  - leading to a parallel class hierarchy:



Spawners

```
class Spawner
{
public:
  virtual ~Spawner() {}
  virtual Monster* spawnMonster() = 0;
};

class GhostSpawner : public Spawner
{
public:
  virtual Monster* spawnMonster()
  {
    return new Ghost();
  }
};

class DemonSpawner : public Spawner
{
public:
  virtual Monster* spawnMonster()
  {
    return new Demon();
  }
};

// You get the idea...
```

- This is obviously not a good way to do this
  - Lots of classes,
  - lots of boilerplate,
  - lots of redundancy,
  - lots of duplication,
  - lots of repeatiton

Solution

- The Prototype pattern offers a solution.
  - The key idea is that an object can spawn other objects similar to itself.
  - If you have one ghost,
    - you can make more ghosts from it.
  - If you have a demon,
    - you can make other demons.
  - Any monster can be treated as a prototypal monster used to generate other versions of itself.

Code

```
class Monster
{
public:
  virtual ~Monster() {}
  virtual Monster* clone() = 0;

  // Other stuff...
};

class Ghost : public Monster {
public:
  Ghost(int health, int speed)
  : health_(health),
    speed_(speed)
  {}

  virtual Monster* clone()
  {
    return new Ghost(health_, speed_);
  }

private:
  int health_;
  int speed_;
};
```

Code

Once all our monsters support that, we no longer need a spawner class for each monster class. Instead, we define a single one:

```
class Spawner
{
public:
  Spawner(Monster* prototype)
  : prototype_(prototype)
  {}

  Monster* spawnMonster()
  {
    return prototype_->clone();
  }

private:
  Monster* prototype_;
};
```

To create a ghost spawner, we create a prototypal ghost instance and then create a spawner holding that prototype:

```
Monster* ghostPrototype = new Ghost(15, 3);
Spawner* ghostSpawner = new Spawner(ghostPrototype);
```

Singleton Pattern

Singleton Pattern

Design Patterns summarizes Singleton like this:

> Ensure a class has one instance, and provide a global point of access to it.

- Sometimes a class works best if there is only one instance of it
    - Consider the file system
    - You don't want multiple instances of the file system class trying to read and write to the hardware at the same time
- The singleton pattern provides a way for a class to ensure at compile time that there is only a single instance of the class.
- It also provides a globally available method to reach this instance

Example

```
class FileSystem
{
public:
  static FileSystem& instance()
  {
    // Lazy initialize.
    if (instance_ == NULL) instance_ = new FileSystem();
    return *instance_;
  }

private:
  FileSystem() {}

  static FileSystem* instance_;
};
```

- Notice that the constructor is private
- The only way to obtain an instance is through the method exposed to public

What is bad about it

- Global access makes it difficult to trace code
- Encourages coupling
- Bad for concurrency

Ask for one, get two

- Singleton provides two things at once:
  - Global access
  - Single instance
- Sometimes you just want one of them
  - A logger must have global access, but you may have multiple instances
  - You want a single instance of a DB connection, but you don't want it to be accessible by everybody

State Pattern

State Pattern

- Consider a piece of code from a platformer:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    yVelocity_ = JUMP_VELOCITY;
    setGraphics(IMAGE_JUMP);
  }
}
```

- What's wrong here?

State Pattern

- Consider a piece of code from a platformer:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    yVelocity_ = JUMP_VELOCITY;
    setGraphics(IMAGE_JUMP);
  }
}
```

- What's wrong here?
  - There's nothing to prevent jumping in the air!

State Pattern

- Consider a piece of code from a platformer:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_)
    {
      isJumping_ = true;
      // Jump...
    }
  }
}
```

- What’s wrong here?
  - There’s nothing to prevent jumping in the air!
- It is fixed.

State Pattern

- Let's add ducking code:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    // Jump if not jumping...
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    setGraphics(IMAGE_STAND);
  }
}
```

- Spot the bug this time?

State Pattern

- Let's add ducking code:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    // Jump if not jumping...
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    setGraphics(IMAGE_STAND);
  }
}
```

- Spot the bug this time?
  - Press down to duck.
  - Press B to jump from a ducking position.
  - Release down while still in the air.

State Pattern

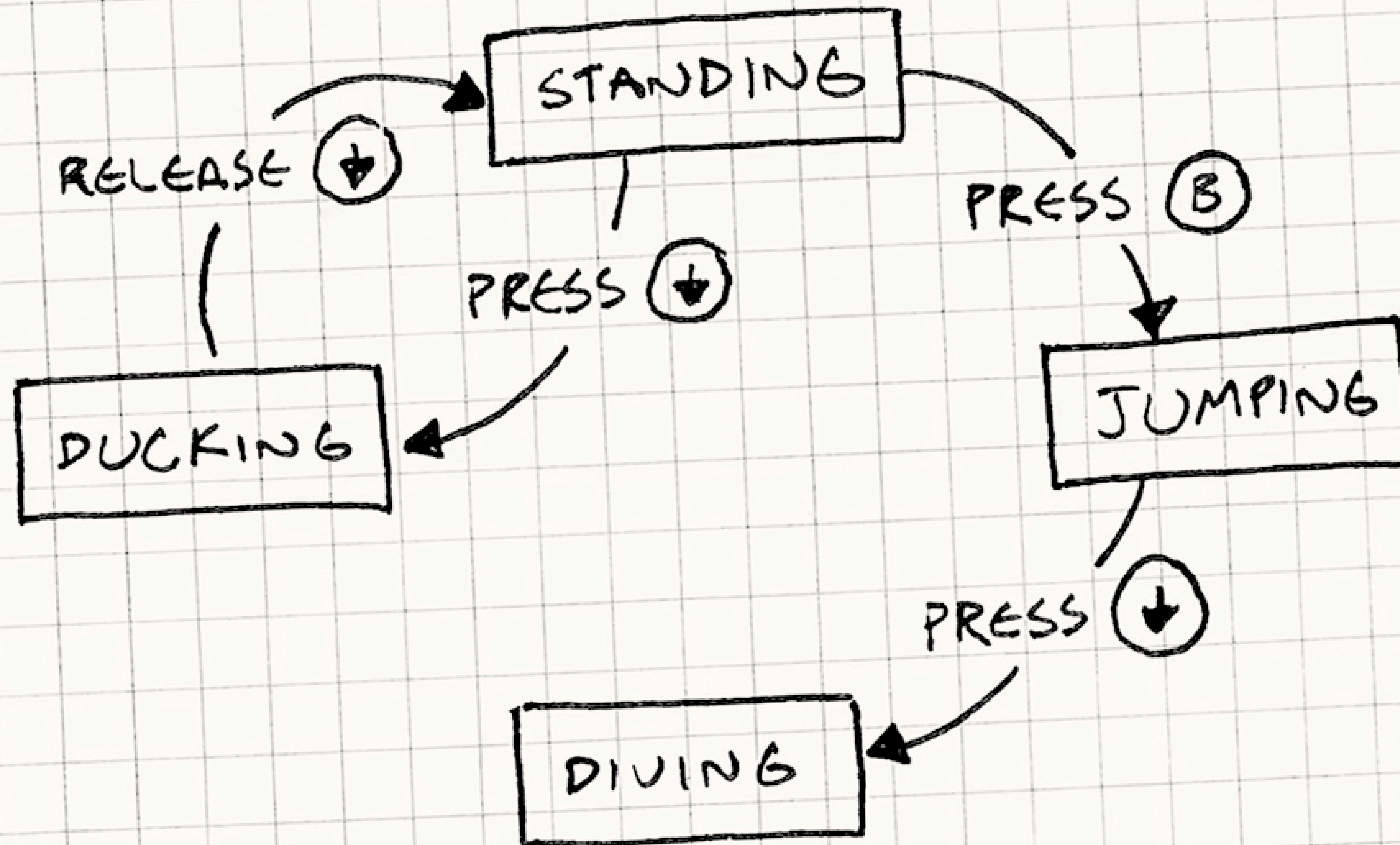
- Let's fix it....again:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_ && !isDucking_)
    {
      // Jump...
    }
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      isDucking_ = true;
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    if (isDucking_)
    {
      isDucking_ = false;
      setGraphics(IMAGE_STAND);
    }
  }
}
```

More actions

- Add walking, diving, shooting, etc. and you are buried deep under bugs.

Finite State Machines



Rules of FSMs

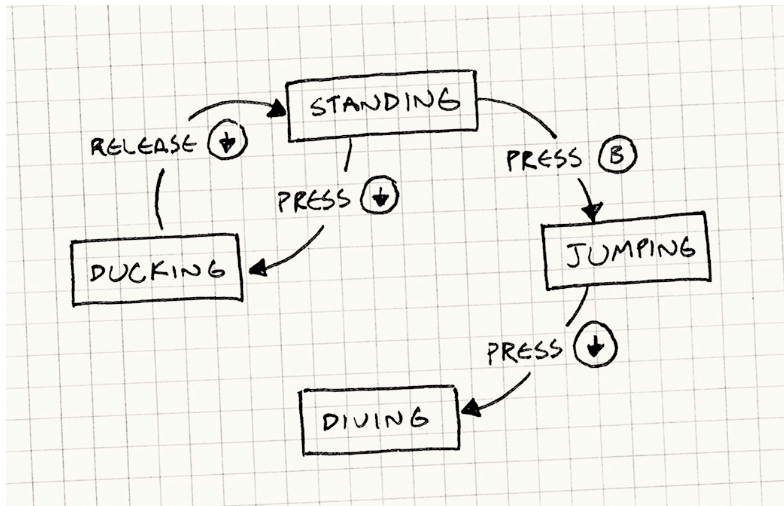
- You have a fixed set of states that the machine can be in.
  - For our example, that's standing, jumping, ducking, and diving.
- The machine can only be in one state at a time.
  - Our heroine can't be jumping and standing simultaneously.
- A sequence of inputs or events is sent to the machine.
- Each state has a set of transitions, each associated with an input and pointing to a state.
  - When an input comes in, if it matches a transition for the current state, the machine changes to the state that transition points to.

Implementation

- Only one state can be active at once
- We will use an enum to represent states:

```
enum State
{
  STATE_STANDING,
  STATE_JUMPING,
  STATE_DUCKING,
  STATE_DIVING
};
```

Implementation




```
void Heroine::handleInput(Input input)
{
  switch (state_)
  {
    case STATE_STANDING:
      if (input == PRESS_B)      {
        state_ = STATE_JUMPING;
        yVelocity_ = JUMP_VELOCITY;
        setGraphics(IMAGE_JUMP);
      }
      else if (input == PRESS_DOWN)      {
        state_ = STATE_DUCKING;
        setGraphics(IMAGE_DUCK);
      }
      break;

    case STATE_JUMPING:
      if (input == PRESS_DOWN)      {
        state_ = STATE_DIVING;
        setGraphics(IMAGE_DIVE);
      }
      break;

    case STATE_DUCKING:
      if (input == RELEASE_DOWN)      {
        state_ = STATE_STANDING;
        setGraphics(IMAGE_STAND);
      }
      break;
  }
}
```

More Problems

We want to add a move where our heroine can duck for a while to charge up and unleash a special attack. While she's ducking, we need to track the charge time.

We add a `chargeTime_` field to `Heroine` to store how long the attack has charged. Assume we already have an `update()` that gets called each frame. In there, we add:

```
void Heroine::update()
{
  if (state_ == STATE_DUCKING)
  {
    chargeTime_++;
    if (chargeTime_ > MAX_CHARGE)
    {
      superBomb();
    }
  }
}
```

More Problems

We need to reset the timer when she starts ducking, so we modify `handleInput()`:

```
void Heroine::handleInput(Input input)
{
  switch (state_)
  {
    case STATE_STANDING:
      if (input == PRESS_DOWN)
      {
        state_ = STATE_DUCKING;
        chargeTime_ = 0;
        setGraphics(IMAGE_DUCK);
      }
      // Handle other inputs...
      break;

      // Other states...
  }
}
```

We had to modify two methods and add a `chargeTime_` field onto `Heroine` even though it's only meaningful while in the ducking state. What we'd prefer is to have all of that code and data nicely wrapped up in one place.

The State Pattern

In the words of the Gang of Four:

> Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

The State Pattern

- First, we define an interface for the state.
- Every bit of behavior that is state-dependent - every place we had a switch before - becomes a virtual method in that interface.
  - For us, that's `handleInput()` and `update()`:

```
class HeroineState
{
public:
  virtual ~HeroineState() {}
  virtual void handleInput(Heroine& heroine, Input input) {}
  virtual void update(Heroine& heroine) {}
};
```

State Classes

For each state, we define a class that implements the interface.

```
class DuckingState : public HeroineState
{
public:
  DuckingState()
  : chargeTime_(0)
  {}

  virtual void handleInput(Heroine& heroine, Input input) {
    if (input == RELEASE_DOWN) {
      // Change to standing state...
      heroine.setGraphics(IMAGE_STAND);
    }
  }

  virtual void update(Heroine& heroine) {
    chargeTime_++;
    if (chargeTime_ > MAX_CHARGE) {
      heroine.superBomb();
    }
  }

private:
  int chargeTime_;
};
```

- We also moved `chargeTime_` out of `Heroine` and into the `DuckingState` class.

State Pattern

Next, we give the `Heroine` a pointer to her current state, lose each big switch, and delegate to the state instead:

```
class Heroine
{
public:
  virtual void handleInput(Input input)
  {
    state_->handleInput(*this, input);
  }

  virtual void update()
  {
    state_->update(*this);
  }

  // Other methods...
private:
  HeroineState* state_;
};
```

State Pattern

Keep the states in the base class and switch as follows:

```
class HeroineState
{
public:
  static StandingState standing;
  static DuckingState ducking;
  static JumpingState jumping;
  static DivingState diving;

  // Other code...
};
```

Each of those static fields is the one instance of that state that the game uses. To make the heroine jump, the standing state would do something like:

```
if (input == PRESS_B)
{
  heroine.state_ = &HeroineState::jumping;
  heroine.setGraphics(IMAGE_JUMP);
}
```